Kdump: Smarter, Easier, Trustier

Vivek Goyal
IBM
vgoyal@in.ibm.com

Neil Horman
Red Hat, Inc.
nhorman@redhat.com

Ken'ichi Ohmichi
NEC Soft
oomichi@mxs.nes.nec.co.jp

Maneesh Soni
IBM
maneesh@in.ibm.com

Ankita Garg
IBM
ankita@in.ibm.com

Abstract

Kdump, a kexec based kernel crash dumping mechanism, has witnessed a lot of new significant development activities recently. Features like, the relocatable kernel, dump filtering, and initrd based dumping have enhanced kdump capabilities are important steps towards making it a more reliable and easy to use solution. New tools like Linux Kernel Dump Test Module (LKDTM) provide an opportunity to automate kdump testing. It can be especially useful for distributions to detect and weed out regressions relatively quickly. This paper presents various newly introduced features and provides implementation details wherever appropriate. It also briefly discusses the future directions, such as early boot crash dumping.

1 Introduction

Kdump is a kernel crash dumping mechanism where a pre-loaded kernel is booted in, to capture the crash dump after a system crash [12]. This pre-loaded kernel, often called as capture kernel, runs from a different physical memory area than the production kernel or regular kernel. As of today, a capture kernel is specifically compiled and linked for a specific memory location, and is shipped as an extra kernel to capture the dump. A relocatable kernel implementation gets rid of the requirement to run the kernel from the address it has been compiled for, instead one can load the kernel at a different address and run it from there. Effectively the distributions and kdump users don't have to build an extra kernel to capture the dump, enhancing ease of use. Section 2 provides the details of relocatable kernel implementation.

Modern machines are being shipped with bigger and bigger RAMs and a high end configuration can possess a tera-byte of RAM. Capturing the contents of the entire RAM would result in a proportionately large core file and managing a tera-byte file can be difficult. One does not need the contents of entire RAM to debug a kernel problem and many pages like userspace pages can be filtered out. Now an open source userspace utility is available for dump filtering and Section 3 discusses the working and internals of the utility.

Currently, a kernel crash dump is captured with the help of init scripts in the userspace in the capture kernel. This approach has some drawbacks. Firstly, it assumes that the root filesystem did not get corrupted and is still mountable in the second kernel. Secondly, minimal work should be done in second kernel and one need not have to run various user space init scripts. This led to the idea of building a custom initial ram disk (initrd) to capture the dump and improve the reliability of the operation. Various implementation details of initrd based dumping are presented in Section 4.

Section 5 discusses the Linux Kernel Dump Test Module (LKDTM), a kernel module, which allows one to set up and trigger crashes from various kernel code paths at run time. It can be used to automate kdump testing procedure to identify bugs and eliminate regressions with lesser efforts. This paper also gives a brief update on device driver hardening efforts in Section 6 and concludes with future work in Section 7.

2 Relocatable bzImage

Generally, the Linux® kernel is compiled for a fixed address and it runs from that address. Traditionally, for i386 and x86_64 architectures, it has been compiled and run from 1MB physical memory location. Later, Eric W. Biederman introduced a config option, `CONFIG_PHYSICAL_START`, which allowed a kernel to be compiled for a different address. This option effectively shifted the kernel in virtual address space and one could compile and run the kernel from a physical address say, 16MB. Kdump used this feature and built a separate kernel for capturing the dump. This kernel was specifically built to run from a reserved memory location.

The requirement of building an extra kernel for dump capturing has not gone over too well with distributions, as they end up shipping an extra kernel binary. Apart from disk space requirements, it also led to increased efforts in terms of supporting and testing this extra kernel. Also from a user's perspective, building an extra kernel is cumbersome.

The solution to the above problem is a relocatable kernel, where the same kernel binary can be loaded and run from a different address than what it has been compiled for. Jan Kratochvil had posted a set of prototype patches to kick- off the discussion on the fastboot mailing list [6]. Later, Eric W. Biederman came up with another set of patches and posted them to LKML for review [8]. Finally, Vivek Goyal picked up Eric's patches, cleaned those up, fixed a number of bugs, incorporated various review comments, and went through multiple rounds of reposting to LKML for inclusion into the mainline kernel. Relocatable kernel implementation is very architecture dependent and support for i386 architecture has been merged with version 2.6.20 of the mainline kernel. Patches for x86_64 have been posted on LKML [11] and are now part of -mm. Hopefully, these will be merged with mainline kernel soon.

2.1 Design Approaches

The following design approaches have been discussed for the relocatable bzImage implementation.

- **Modify kernel text/data mapping at run time**
  At run time, the kernel determines where it has been loaded by the boot-loader and it updates its page tables to reflect the right mapping between kernel virtual and physical addresses for kernel text and data. This approach has been adopted for the x86_64 implementation.
- **Relocate using relocation information**
  This approach forces the linker to generate relocation information. These relocations are processed and packed into the bzImage. The uncompressed kernel code decompresses the kernel, performs the relocations, and transfers control to the protected mode kernel. This approach has been adopted by the i386 implementation.

2.2 Design Details (i386)

In i386, kernel text and data are part of linearly mapped region which has got hard-coded assumptions about virtual to physical address mapping. Hence, it is probably difficult to adopt the modifying the page table approach for implementing a relocatable kernel. Instead, a simpler, non-intrusive approach is to ask the linker to generate relocation information, pack this relocation information into bzImage, and the uncompressed kernel code can process these relocations before jumping to the 32-bit kernel entry point (`startup_32()`).

2.2.1 Relocation Information Generation

Relocation information can be generated in many ways. The initial experiment was to compile the kernel as shared object file (`-shared`) which generated the relocation entries. Eric had posted the patches for this approach [9] but it was found that the linker also generated the relocation entries for absolute symbols (for some historical reason) [1]. By definition, absolute symbols are not to be relocated, but, with this approach, absolute symbols also ended up being relocated. Hence this method did not prove to be a viable one.

Later, a different approach was taken where the i386 kernel is built with the linker option `--emit-relocs`. This option builds an executable vmlinux and still retains relocation information in a fully linked executable. This increases the size of vmlinux by around 10%, though this information is discarded at runtime. The kernel build process goes through these relocation entries and filters out PC relative relocations, as these don't have to be adjusted if bzImage is loaded at a different physical address. It also filters out the relocations generated with respect to absolute symbols because absolute symbols don't have to be relocated. The rest of the relocation offsets are packed into the compressed vmlinux. Figure 1 depicts the new i386 bzImage build process.

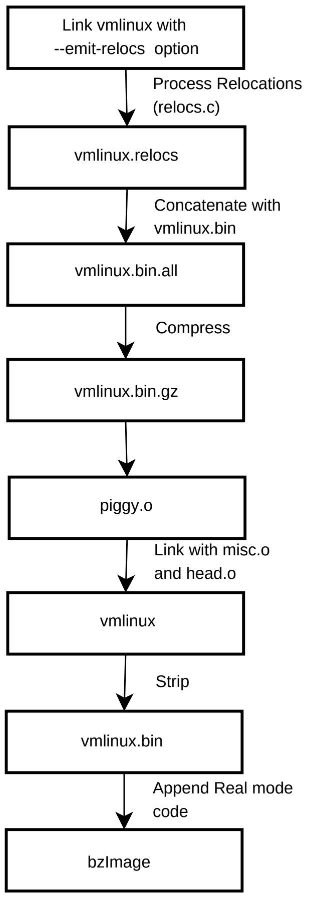
Link vmlinux with --emit-relocs option
Process Relocations (relocs.c)
vmlinux.relocs
Concatenate with vmlinux.bin
vmlinux.bin.all
Compress
vmlinux.bin.gz
piggy.o
Link with misc.o and head.o
vmlinux
Strip
vmlinux.bin
Append Real mode code
bzImage

Figure 1: i386 bzImage build process

2.2.2 In-place Decompression

The code for decompressing the kernel has been changed so that decompression can be done in-place. Now the kernel is not first decompressed to any other memory location then merged and put at final destination and there are no hard-coded assumptions about the intermediate location [7]. This allows the kernel to be decompressed within the bounds of its uncompressed size and it will not overwrite any other data. Figure 2 depicts the new bzImage decompression logic.

At the same time, the decompressor is compiled as position independent code (-fPIC) so that it is not bound to a physical location, and it can run from anywhere. This code has been carefully written to make sure that it runs even if no relocation processing is done.

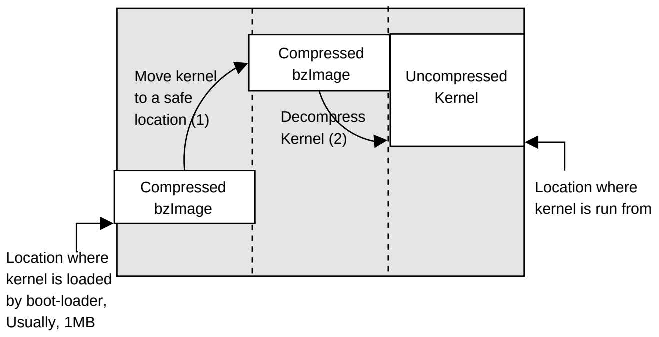
Move kernel to a safe location (1)
Compressed bzImage
Decompress Kernel (2)
Uncompressed Kernel
Compressed bzImage
Location where kernel is run from
Location where kernel is loaded by boot-loader, Usually, 1MB

Figure 2: In-place bzImage decompression

2.2.3 Perform Relocations

After decompression, all the relocations are performed. Uncompressed code calculates the difference between the address for which the kernel was compiled and the address at which it is loaded. This difference is added to the locations as specified by relocation offsets and control is transferred to the 32-bit kernel entry point.

2.2.4 Kernel Config Options

Several new config options have been introduced for relocatable kernel implementation. `CONFIG_RELOCATABLE` controls whether the resulting bzImage is relocatable or not. If this option is not set, no relocation information is generated in the vmlinux.

Generally, bzImage decompresses itself to the address it has been compiled for (`CONFIG_PHYSICAL_START`) and runs from there. But if `CONFIG_RELOCATABLE` is set, then it runs from the address it has been loaded at by the boot-loader and it ignores the compile time address.

`CONFIG_PHYSICAL_ALIGN` option allows a user to specify the alignment restriction on the physical address the kernel is running from.

2.3 Design Details (x86_64)

In x86_64, kernel text and data are not part of the linearly mapped region and are mapped in a separate 40MB virtual address range. Hence, one can easily remap the kernel text and data region depending on where the kernel is loaded in the physical address space.

The kernel decompression logic has been changed to do an in-place decompression. The changes are similar to those as discussed for i386 architecture.

2.3.1 Kernel text And data Mapping Modification

Normally, the kernel text/data virtual and physical addresses differ by an offset of `__START_KERNEL_map` (0xffffffff80000000UL). At run time, this offset will change if the kernel is not loaded at the same address it has been compiled for. Kernel initial boot code determines where it is running from and it updates its page tables accordingly. This shift in address is calculated at run time and is stored in a variable `phys_base`. Figure 3 depicts the various mappings.

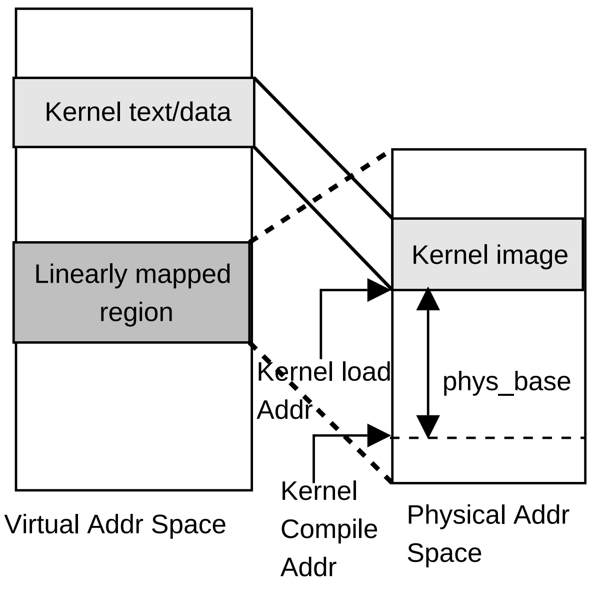

Figure 3: x86_64 kernel text/data mapping update

2.3.2 __pa_symbol() and __pa() Changes

Given the fact kernel text/data mapping changes at run time, some `__pa()`-related macro definitions need to be modified.

As mentioned previously, the kernel determines the difference between the address it has been compiled for and the address it has been loaded at and stores that shift in the variable `phys_base`.

Currently, `__pa_symbol()` is used to determine the physical address associated with a kernel text/data virtual address. Now this mapping is not fixed and can vary at run time. Hence, `__pa_symbol()` has been updated to take into the account the offset `phys_base` while calculating the physical address associated with a kernel text/data area.

```
#define __pa_symbol(x)      \
  ({unsigned long v;  \
    asm("" : "=r" (v) : "0" (x));\
    ((v - __START_KERNEL_map) + phys_base);})
```

`__pa()` should be used only for virtual addresses belonging to a linearly mapped region. Currently, this macro can map both the linearly mapped region and the kernel/text data region. But, now it has been updated to map only the kernel linearly mapped region, keeping in line with the rest of the architectures. As the kernel linearly mapped region mappings don't change because of kernel image location, `__pa()` does not have to handle the kernel load address shift (`phys_base`).

```
#define __pa(x) \
    ((unsigned long)(x) - PAGE_OFFSET)
```

2.3.3 Kernel Config Options

It is similar to i386, except that there is no option `CONFIG_PHYSICAL_ALIGN` and alignment is set to 2MB.

2.4 bzImage Protocol Extension

The bzImage protocol has been extended to communicate relocatable kernel information to the boot-loader. Two new fields, `kernel_alignment` and `relocatable_kernel`, have been added to the bzImage header. The first one specifies the physical address alignment requirement for the protected mode kernel, and the second one indicates whether this protected mode kernel is relocatable or not.

A boot-loader can look at the `relocatable_kernel` field and decide if the protected mode component should be loaded at the hard-coded 1MB address or it can be loaded at other addresses too. Kdump uses this feature and kexec boot-loader loads the relocatable bzImage at non-1MB address, in a reserved memory area.

3 Dump Filtering

On modern machines with a large amount of memory, capturing the contents of the entire RAM could create a huge dump file, which might be in the range of terabytes. It is difficult to manage such a huge dump file, both while storing it locally and while sending it to a remote host for analysis. *makedumpfile* is a userspace utility to create a smaller dump file by dump filtering, or by compressing the dump data, or both [5].

3.1 Filtering Options

Often, many pages like userspace pages, free pages, and cache pages might not be of interest to the engineer analyzing the crash dump. Hence, one can choose to filter out those pages. The following are the types of pages one can filter out:

- Pages filled with zero
  `makedumpfile` distinguishes this page type by reading each page. These pages are not part of the dump file but the analysis tool is returned zeros while accessing the filtered zero page.

- Cache pages
  `makedumpfile` distinguishes this page type by reading the members `flags` and `mapping` in `struct page`. If both the `PG_lru` bit and `PG_swapcache` bit of `flags` are on and `PAGE_MAPPING_ANON` bit of `mapping` is off, the page is considered to be a cache page.

- User process data pages
  `makedumpfile` distinguishes this page type by reading the member `mapping` in `struct page`. If `PAGE_MAPPING_ANON` bit of `mapping` is on, the page is considered to be a user process data page.

- Free pages
  `makedumpfile` distinguishes this page type by reading the member `free_area` in `struct zone`. If the page is linked into the member `free_area`, the page is considered to be a free page.

3.2 Filtering Implementation Details

`makedumpfile` examines the various memory management related data structures in the core file to distinguish between page types. It uses the crashed kernel vmlinux, compiled with debug information, to retrieve a variety of information like data structure size, member offset, symbol addresses, and so on.

The memory management information depends on the Linux version, the architecture of the processor, and the memory model (`FLATMEM`, `DISCONTIGMEM`, `SPARSEMEM`). For example, the symbol name of `struct pglist_data` is `node_data` on linux-2.6.18 x86_64 `DISCONTIGMEM`, but it is `pgdat_list` on linux-2.6.18 ia64 `DISCONTIGMEM`. `makedumpfile` supports these varieties.

To begin with, `makedumpfile` infers the memory model used by the crashed kernel, by searching for the symbol `mem_map`, `mem_section`, `node_data`, or `pgdat_list` in the binary file of the production kernel. If symbol `mem_map` is present, the crashed kernel used `FLATMEM` memory model or if `mem_section` is present, the crashed kernel used `SPARSEMEM` memory model or if `node_data` or `pgdat_list` is present, the crashed kernel used `DISCONTIGMEM` memory model.

Later it examines the `struct page` entry of each page frame and retrieves the members `flags` and `mapping`. The size of `struct page` and the member field offsets are extracted from the `.debug_info` section of the debug-compiled vmlinux of the production kernel. Various symbol virtual addresses are retrieved from the symbol table of production kernel binary.

The organization of `struct page` entry arrays, depends on the memory model used by the kernel. For the `FLATMEM` model on linux-2.6.18 i386, `makedumpfile` determines the virtual address of the symbol `mem_map` from vmlinux. This address is translated into file offset with the help of `/proc/vmcore` ELF headers and finally it reads the `mem_map` array at the calculated file offset from core file. Other page types in various memory models are distinguished in similar fashion.

3.3 Dump File Compression

The dump file can be compressed using standard compression tools like gzip to generate smaller footprint. The only drawback is that one will have to uncompress the whole file before starting the analysis. Alternatively, one can compress individual pages and decompress a particular page only when analysis tool accesses it. This reduces the disk space requirement while analyzing a crash dump.

`makedumpfile` allows for the creation of compressed dump files where compression is done on a per page basis. `diskdump` has used this feature in the past and `makedumpfile` has borrowed the idea [3].

3.4 Dump File Format

By default, `makedumpfile` creates a dump file in the kdump-compressed format. It is based on `diskdump` file format with minor modifications. The `crash` utility can analyze kdump-compressed format.

`makedumpfile` can also create a dump file in ELF format which can be opened by both `crash` and `gdb`. The ELF format does not support compressed dump files.

3.5 Sample Dump Filtering Results

The dump file size depends on the production kernel's memory usage. Tables 1 and 2 show the dump file size reduction in two possible cases. In the first table, most of the production kernel's memory is free, as dump was captured immediately after a system boot and filtering out free pages is effective. In the second table, most of the memory is used as cache, as a huge file was being decompressed while dump was captured, and filtering out cache pages is effective.

linux-2.6.18, x86_64 Memory:5GB	
Filtering option	Size Reduction
Pages filled with zero	76%
Cache pages	16%
User process data pages	1%
Free pages	78%
All the above types	97%

Table 1: Dump filtering on system containing many free pages

linux-2.6.18, x86_64 Memory:5GB	
Filtering option	Size Reduction
Pages filled with zero	3%
Cache pages	91%
User process data pages	1%
Free pages	1%
All the above types	94%

Table 2: Dump filtering on system containing many cache pages

4 Kdump initramfs

In the early days of kdump, crash dump capturing was automated with the help of init scripts in userspace. This approach was simple and easy, but it assumed that the root filesystem was not corrupted during system crash and could still be mounted safely in the second kernel. Another consideration is that one should not have to run various other init scripts before he/she starts saving the dump. Other scripts unnecessarily consume precious kernel memory and possibly can lead to reduced reliability.

These limitations led to the idea of capturing the crash dump from early userspace (initial ramdisk context). Saving the dump before even a single init script runs, probably adds to the reliability of the operation and precious memory is not consumed by un-required init scripts. Also, one could specify a dump device other than the root partition, which is guaranteed to be safe.

A prototype implementation of initrd based dumping was available in Fedora® 6. This was a basic scheme implemented along the lines of nash shell based standard boot initrd and had various drawbacks like bigger ramdisk size, limited dump destination devices supported, and limited error handling capability because of constrained scripting environment.

The above limitations triggered the redesign of the initrd based dumping mechanism. The following sections provide the details of the new design and also highlight the short-comings of the existing implementation, wherever appropriate.

4.1 Busybox-based Initial RAM Disk

Initial implementation of initrd based dumping was roughly based on the initramfs files generated by the `mkinitrd` utility. The newer design, uses Busybox [2] utilities to generate the kdump initramfs. The advantages of this scheme become evident in the following discussion.

4.1.1 Managing the initramfs Size

One of the primary design goals was to keep the initramfs as small as possible for two reasons. First, one wants to reserve as little memory as possible for crash dump kernel to boot and second, out of the reserved memory, one wants to keep the free memory pool as large as possible, to be used by kernel and drivers.

Initially it was considered to implement all of the required functionality for kdump in a statically linked binary, written in C. This binary would have been smaller than Busybox, as it would avoid inclusion of unused Busybox bits. But maintainability of the above approach was a big concern, keeping in mind the vast array of functionality it had to support. The feature list included the ability to copy files to nfs mounts, to local disk drives, to local raw partitions, and to remote servers via ssh.

Upon a deeper look, the problem space resembled more and more that of an embedded system which made the immediate solution to many of the constraints self evident: Busybox [2].

Following are some of the utilities which are typically packed into the initial ramdisk and contribute to the size bloat of initramfs.

- `nash`: A non-interactive shell-like environment
- The `cp` utility
- The `scp` and `ssh` utilities: If a scp remote target is selected
- The `ifconfig` utility
- The `dmsetup` and `lvm` utilities: For software raided and lvm file systems

Some of these utilities are already built statically. However, even if one required the utilities to be dynamically linked, various libraries have to be pulled in to satisfy dependencies and the initramfs image size skyrockets. In the case of the earlier initramfs for kdump, depending on the configuration, the inclusion of `ssh`, `scp`, `ifconfig`, and `cp` required the inclusion of the following libraries:

libacl.so.1	libz.so.1
libselinux.so.1	libnsl.so.1
libc.so.6	libcrypt.so.1
libattr.so.1	libgssapi_krb5.so.2
libdl.so.2	libkrb5.so.3
libsepol.so.1	libk5crypto.so.3
linux-gate.so.1	libcom_err.so.2
libresolv.so.2	libkrb5support.so
libcrypto.so.6	ld-linux.so.2
libutil.so.1	

Given these required utilities and libraries, the initramfs was initially between 7MB and 11MB uncompressed, which seriously cut into the available heap presented to the kernel and the userspace applications which needed it during the dump recovery process.

Busybox immediately provided a remedy to many of the size issues. By using Busybox, the cp and ifconfig utilities were no longer needed, and with them went away the need for most of the additional libraries. With Busybox, our initramfs size was reduced to a range of 2MB to 10MB.

4.1.2 Enhanced Flexibility

A Busybox based initramfs implementation vastly increased kdump system flexibility. Initially, the nash interpreter allowed us a very small degree of freedom in terms of how we could capture crash dumps. Given that nash is a non-interactive script interpreter with an extremely limited conditional handling infrastructure, we were forced in our initial implementation to determine, at initramfs creation time, exactly what our crash procedure would be. In the event of any malfunction, there was only one error handling path to choose, which was failing to capture the dump and rebooting the system.

Now, with Busybox, we are able to replace nash with any of the supported Busybox shells (msh was chosen, since it was the most bash-like shell that Fedora's Busybox build currently enables). This switch gave us several improvements right away, such as an increase in our ability to make decisions in the init script. Instead of a script that was in effect a strictly linear progression of commands, we now had the ability to create if-then-else conditionals and loops, as well as the ability to create and reference variables in the script. We could now write an actual shell script in the initramfs, which allowed us to, among many other things, to recover from errors in a less drastic fashion. We now have the choice to do something other than simply reboot the system and lose the core file. Currently, it tries to mount the root filesystem and continue the boot process, or drop to an interactive shell within the initramfs so that a system administrator can attempt to recover the dump manually.

In fact, the switch to Busybox gave us a good deal of additional features that we were able to capitalize on. Of specific note was the additional networking ability in Busybox. The built-in ifup/ifdown network interface framework allowed us to bring network interfaces up and down easily, while the built-in vconfig utility allowed us to support remote core capture over vlan interfaces. Furthermore since we now had a truly scriptable interface, we were able to use the sysfs interface to enslave interfaces to one another, emulating the ifenslave utility, which in turn allows us to dump cores over bonded interfaces. Through the use of sysfs, we are also able to dynamically query the devices that are found at boot time and create device files for them on the fly, rather than having to anticipate them at initramfs creation time. Add to that the ability to use Busybox's findfs utility to identify local partitions by disklabel or uuid, and we are able to dynamically determine our dump location at boot time without needing to undergo a kdump initramfs rebuild every time local disk geometry changes.

4.2 Future Goals

In the recent past, our focus in terms of kdump userspace implementation has been on moving to Busybox in an effort to incorporate and advance upon the functionality offered by previous dump capture utilities, while minimizing the size impact of the initramfs and promoting maintainability. Now that we have achieved these goals, at least in part, our next set of goals include the following:

- **Cleanup initramfs generation** – The generation of the initramfs has been an evolutionary process. Current initramfs generation script is a heavily modified version of its predecessor to support the use of Busybox. This script needs to be re-implemented to be more maintainable.

- **Config file formalization** – The configuration file syntax for kdump is currently very ad hoc, and does not easily support expansion of configuration directives in any controlled manner. The configuration file syntax should be formalized.

- **Multiple dump targets** – Currently, the initramfs allows the configuration of one dump target, and a configurable failure action in the event the dump capture fails. Ideally, the configuration file should support the listing of several dump targets as alternatives in case of failures.

- **Further memory reduction** – While we have managed to reduce memory usage in the initramfs by a large amount, some configurations still require the use of large memory footprint binaries (most notably scp and ssh). Eventually, we hope to switch to using a smaller statically linked ssh client for use in remote core capture instead, to reduce the top end of our memory usage.

5 Linux Kernel Dump Test Module

Before adopting any dumping mechanism, it is important to ascertain that the solution performs reliably in most crash scenarios. To achieve this, one needed a tool which can be used to trigger crash dumps from various kernel code paths without patching and rebuilding the kernel. LKDTM (Linux Kernel Dump Test Module) is a dynamically loadable kernel module, that can be used for forcing a system crash in various scenarios and helps in evaluating the reliability of a crash dumping solution. It has been merged with the mainline kernel and is available in kernel version 2.6.19.

LKDTM is based on LKDTT (Linux Kernel Dump Test Tool) [10], but has an entirely different design. LKDTT inserts the crash points statically and one must patch and rebuild the kernel before it can be tested. On the other hand, LKDTM makes use of `jprobes` infrastructure and allows crash points to be inserted dynamically.

5.1 LKDTM Design

LKDTM artificially induces system crashes at predefined locations and triggers dump for correctness testing. The goal is to widen the coverage of the tests, to take into account the different conditions in which the system might crash, for example, the state of the hardware devices, system load, and context of execution.

LKDTM achieves the crash point insertion by using jumper probes (`jprobes`), the dynamic kernel instrumentation infrastructure of the Linux kernel. The module places a *jprobe* at the entry point of a critical function. When the kernel control flow reaches the function, as shown in Figure 4, the probe causes the registered helper function to be called before the actual function is executed. At the time of insertion, each crash point is associated with two attributes: the action to be triggered and the number of times the crash point is to be hit before triggering the action (similar to LKDTT). In the helper function, if it is determined that the count associated with the crash point has been hit, the specified action is performed. The supported action types, referred to as Crash Types, are *kernel panic, oops, exception, and stack overflow*.

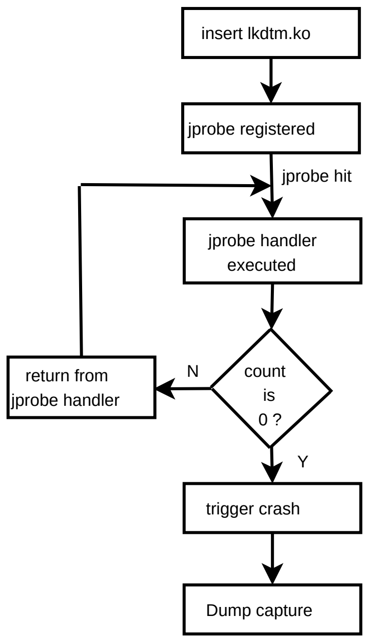

Figure 4: LKDTM functioning

`jprobes` was chosen over `kprobes` in order to ensure that the action is triggered in the same context as that of the critical function. In the case of `kprobes`, the helper function is executed in the *int 3* trap context. Whereas, when the *jprobe* is hit, the underlying `kprobes` infrastructure points the saved instruction pointer to the *jprobe's* handler routine and returns from the int3 trap (refer Documentation/kprobes.txt for the working of `kprobes/jprobes`). The helper routine is then executed in the same context as that of the critical function, thus preserving the kernel's execution mode.

5.2 Types of Crash Points

The basic crash points supported by LKDTM are same as supported by LKDTT. These are as follows:

IRQ handling with IRQs disabled (`INT_HARDWARE_ENTRY`) The *jprobe* is placed at the head of the function `__do_IRQ`, which processes interrupts with IRQs disabled.

IRQ handling with IRQs enabled (`INT_HW_IRQ_EN`) The *jprobe* is placed at the head of the function `handle_IRQ_event`, which processes interrupts with IRQs enabled.

Tasklet with IRQs disabled (`TASKLET`) This crash point recreates crashes that occur when the tasklets are being executed with interrupts disabled. The *jprobe* is placed at the function `tasklet_action`.

Block I/O (`FS_DEVRW`) This crash point crashes the system when the filesystem accesses the low-level block devices. It corresponds to the function `ll_rw_block`.

Swap-out (`MEM_SWAPOUT`) This crash point causes the system to crash while in the memory swapping is being performed.

Timer processing (`TIMERADD`) The *jprobe* is placed at function `hrtimer_start`.

SCSI command (`SCSI_DISPATCH_CMD`) This crash point is placed in the SCSI dispatch command code.

IDE command (`IDE_CORE_CP`) This crash point brings down the system while handling I/O on IDE block devices.

New crash points can be added, if required, by making changes to drivers/misc/lkdtm.c file.

5.3 Usage

The LKDTM module can be built by enabling the `CONFIG_LKDTM` config option under the `Kernel hacking` menu. It can then be inserted into the running kernel by providing the required command line arguments, as shown:

```
#modprobe lkdtm cpoint_name=<> cpoint_
type=<> [cpoint_count={>0}] [recur_
count={>0}]
```

5.4 Advantages/disadvantages of LKDTM

LKDTT has kernel space and user space components. In order to make use of LKDTT, one has to apply the kernel patch and rebuild the kernel. Also, it makes use of the Generalised Kernel Hooks Interface (GHKI), which is not part of the mainline kernel. On the other hand, using LKDTM is extremely simple and is merged into mainline kernels. The crash point can be injected into a running kernel by simply inserting the kernel module.

The only shortcoming of LKDTM is that the crash point cannot be placed in the middle of the function without changing the context of execution, unlike LKDTT.

5.5 Kdump Testing Automation

So far kdump testing was done manually but it was difficult and very time consuming process. Now it has been automated with the help of LKDTM infrastructure and some scripts.

LTP (Linux Test Project) seems to be the right place for such testing automation framework. A patch has been posted to LTP mailing list [4]. This should greatly help distributions in quickly identify regressions with every new release.

These scripts set up a cron job which starts on a reboot and inserts either LKDTM or an elementary testing module called `crasher`. Upon a crash, a crash dump is automatically captured and saved to a pre-configured location. This is repeated for various crash points as supported by LKDTM. Later, these scripts also open the captured dump and do some basic sanity verification.

The tests can be started by simply executing the following from within the tarball directory:

```
# ./setup
# ./master run
```

The detailed instructions on the usage have been documented in the README file, which is part of the tarball.

6 Device Driver Hardening

Device driver initialization in a capture kernel continues to be a pain point. Various kinds of problems have been reported. A very common problem is the pending messages/interrupts on the device from previous the kernel's context. This interrupt is delivered to the driver in the capture kernel's context and it often crashes because of state mismatch. A solution is based on the fact that the device should have a way to allow the driver to reset it. Reset should bring it to a known state from where the driver can continue to initialize the device.

PCI bus reset can probably be of help here, but it is uncertain how the PCI bus can be reset from software. There does not seem to be a generic way, but PowerPC® firmware allows doing a software reset of the PCI buses and the Extended Error Handling (EEH) infrastructure makes use of it. We are looking into using EEH functionality to reset the devices while the capture kernel boots.

Even if there is a way to reset the device, device drivers might not want to reset it all the time as resetting is generally time consuming. To resolve this issue, a new command line parameter `reset_devices` has been introduced. When this parameter is passed on the command line, it is an indication to the driver that it should first try to reset the underlying device and then go ahead with the rest of the initialization.

Some drivers like megaraid, mptfusion, ibmvscsi and ibmveth reported issues and have been fixed. MPT tries to reset the device if it is not in an appropriate state and megaraid sends a FLUSH/ABORT message to the device to flush all the commands sent from previous kernel's context. More problems have been reported with aacraid and cciss drivers which are yet to be fixed.

7 Early Boot Crash Dumping

Currently, kdump does not work if the kernel crashes before the boot process is completed. For kdump to work, the kernel should be booted up so that the new kernel is pre-loaded. During the development phase, many developers run into issues when the kernel is not booting at all, and making crash dumps work in those scenarios will be useful.

One idea is to load the kernel from early userspace (initrd/initramfs), but that does not solve the problem entirely because the kernel can crash earlier than that.

Another possibility is to use the `kboot` boot-loader to pre-load a dump capture kernel in the memory somewhere and then launch the production kernel. This production kernel will jump to the already loaded capture kernel in case of a boot time crash. Figure 5 illustrates the above design.

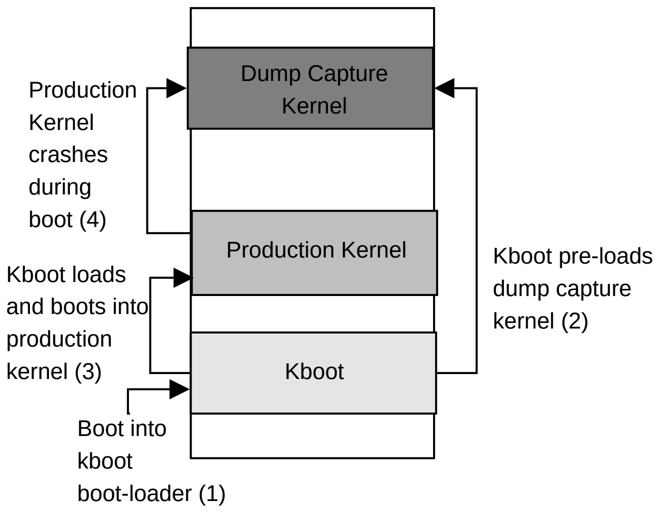
Dump Capture Kernel
Production Kernel crashes during boot (4)
Production Kernel
Kboot loads and boots into production kernel (3)
Kboot pre-loads dump capture kernel (2)
Kboot
Boot into kboot boot-loader (1)

Figure 5: Early boot crash dumping

8 Conclusions

Kdump has come a long way since the initial implementation was merged into the 2.6.13 kernels. Features like the relocatable kernel, dump filtering, and initrd-based dumping have made it an even more reliable and easy to use solution. Distributions are in the process of merging these features in upcoming releases for mass deployment.

The only problem area is the device driver initialization issues in the capture kernel. Currently, these issues are being fixed on a per-driver basis when they are reported. We need more help from device driver maintainers to fix the reported issues. We are exploring the idea of performing device reset using EEH infrastructure on Power and that should further improve the reliability of kdump operation.

9 Legal Statement

References

[1] Absolute symbol relocation entries. `http://lists.gnu.org/archive/html/bug-binutils/2006-07/msg00080.html`.

[2] Busybox. `http://www.busybox.net/`.

[3] diskdump project. `http://sourceforge.net/projects/lkdump`.

[4] Ltp kdump test suite patch tarball. `http://marc.theaimsgroup.com/?l=ltp-list&m=117163521109921&w=2`.

[5] makedumpfile project. `https://sourceforge.net/projects/makedumpfile`.

[6] Jan Kratochvil's prototype implementation. `http://marc.info/?l=osdl-fastboot&m=111829071126481&w=2`.

[7] Werner Almsberger. Booting Linux: The History and the Future. In *Proceedings of the Linux Symposium, Ottawa*, 2000.

[8] Eric W. Biederman. Initial prototype implementation. `http://marc.info/?l=linux-kernel&m=115443019026302&w=2`.

[9] Eric W. Biederman. Shared library based implementation. `http://marc.info/?l=osdl-fastboot&m=112022378309030&w=2`.

[10] Fernando Luis Vazquez Cao. Evaluating Linux Kernel Crash Dumping Mechanisms. In *Proceedings of the Linux Symposium, Ottawa*, 2005.

[11] Vivek Goyal. x86-64 relocatable bzimage patches. `http://marc.info/?l=linux-kernel&m=117325343223898&w=2`.

[12] Vivek Goyal. Kdump, A Kexec Based Kernel Crash Dumping Mechanism. In *Proceedings of the Linux Symposium, Ottawa*, 2005.

Proceedings of the Linux Symposium

Volume One

June 27th–30th, 2007
Ottawa, Ontario
Canada